Center for Space Mission Information and Software Systems

Re-engineering Legacy Mission Software Systems

Charles D. Norton and Viktor K. Decyk

High Performance Computing Systems and Applications Group
Center for Space Mission Information and Software Systems
Imaging and Spectrometry Systems Technology Section – 385

Final Report
September 15, 2000

National Aeronautics and Space Administration
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109-8099

Table of Contents

Contact

Dr. Charles D. Norton
NASA Jet Propulsion Laboratory
California Institute of Technology
MS 168-522
4800 Oak Grove Drive
Pasadena, CA 91109-8099
Email: Charles.D.Norton@jpl.nasa.gov

Dr. Viktor K. Decyk
University of California at Los Angeles
Department of Physics and Astronomy
Knudsen 1-130C
Los Angeles, CA 90095-1547
Email: decyk@physics.ucla.edu

Acknowledgment

This work was developed at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration and supported by a grant from the Jet Propulsion Laboratory Center for Space Mission Information and Software Systems Software Engineering Technology Work Area.

Web Information

http://www.csmiss.jpl.nasa.gov/
http://hpc.jpl.nasa.gov/PEP/nortonc/csmiss.html

1 Overview of Research and Results

JPL missions are constantly evolving, increasing the demands imposed on critical software systems and applications. This ambition requires more modern, flexible, reusable, and extensible component-based software that does not abandon the production applications required for success. Resolving this problem is vital to the future of software integration with mission planning at JPL.

In this report a general software design methodology is presented for modernization of legacy scientific and engineering applications. This technique leverages the existing investment in production codes while infusing modern software engineering principles. Furthermore, various classes of mission flight software throughout JPL, such as flight, data, and analysis systems software, will be discussed to determine how our technology could benefit such systems.

These ideas have been successfully applied to the *Modeling and Analysis for Controlled Optical Systems* (MACOS) software important to the Next Generation Space Telescope (NGST), Space Interferometry Mission (SIM), and other projects. MACOS, developed at JPL, provides tools for analysis of optical systems, model generation, and modeling for system-level design and analysis tasks.

The modernized software has been delivered to the MACOS project. Also discussions were held with members of the *Mission Data System* (MDS) project to determine how legacy software issues may affect their goal of redesigning how spacecraft are conceived, designed, launched, and controlled. To learn more about the software tools becoming available for modernization we visited *Boeing Phantom Works* for a technical interchange. Boeing has applied the use of automatically generated software wrappers to help new software interact with legacy software for embedded flight systems. We have also identified other tools in the market that may assist in partial automation of our methodology. In addition, meetings with the *Navigation and Flight Mechanics Section* have prompted new work involving modernizing their Mars navigation software.

Finally, aspects of our work have been presented at the Workshop on OpenMP Applications and Tools Conference held at the San Diego Supercomputing Center, Lawrence Livermore National Laboratory, Boeing Phantom Works, Navigation Software Development Group (312), and the JPL IT Leadership Council. Also, a paper we submitted on modernization techniques for legacy scientific software has been accepted for oral presentation at the Conference on Computational Physics 2000.

This report summarizes our activities under the CMISSS Software Engineering Technology Work Area. A more detailed tutorial guide on our modernization techniques will be presented in a separate publication.

2 Modernizing Scientific Software

Legacy software has great value since it is generally well debugged, produces results that are trusted, and is actively meeting end-user goals. The amount of hidden expert knowledge embedded in such software can be significant making its preservation important. Nevertheless, legacy software has limitations. It can be difficult to extend, modify, and it does not support collaborative development very well. This can impede the ability to meet new and expanded mission goals as timelines and budgets become tighter. One approach to this problem is to rewrite the software from scratch, but this may introduce more serious costs. In particular, developing new verification and validation tests can be expensive. Also, ensuring that the legacy code was faithfully rewritten, regardless of the programming language applied, cannot always be guaranteed.

Generally, if the functionality of the legacy software is sound, it can be wrapped in a modern interface where the original code is mostly unmodified. The idea of wrapping code means that the original legacy software is preserved while a new layer of software is introduced to separate the old software from the new software. The wrapper provides the best means of retaining the functionality of the legacy software investment while providing a more flexible context from which new software, based on modern concepts, can be introduced.

There are many benefits to this approach:

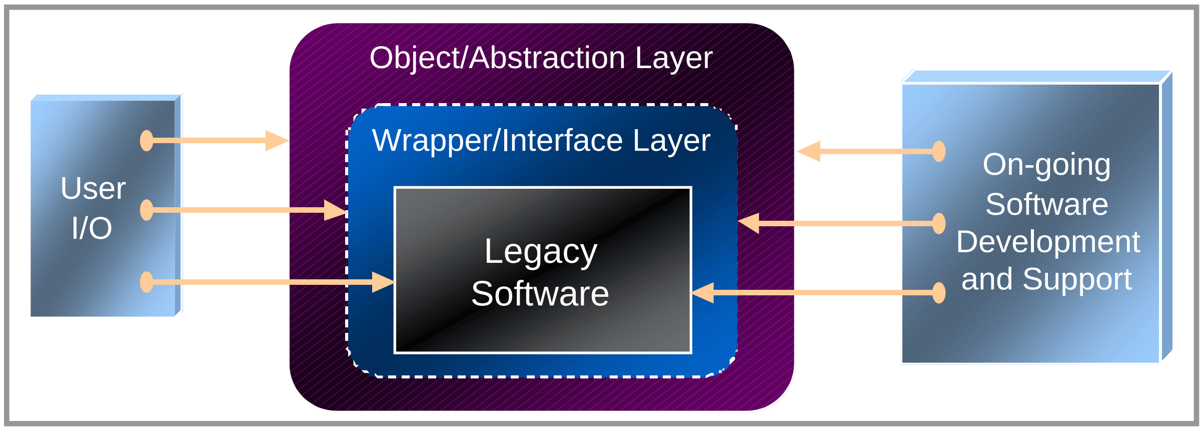
1. Software remains in productive use while applications are modernized.
2. Avoids costly and potentially harmful software rewrites.
3. Promotes collaborative development while resolving organization problems exhibited in older codes.
4. Re-engineering occurs more quickly than rewriting, while preserving verification and validation tests, especially when the original programmers are involved.
5. Old bugs are uncovered.

Extending the functionality of legacy systems has become more important as modern applications increase in complexity and require the interaction of multiple contributors.

2.1 The Technology Applied

We have found that Fortran 90/95 has new features to support object-oriented principles beneficial for scientific programming and introduce a design methodology that defines a step-by-step process to modernize legacy application codes using state-of-the art software practices. While we emphasize Fortran applications, due to the abundance of Fortran legacy codes, similar techniques can be applied to software written in other domain-specific or general-purpose languages, including C or C++. As such, existing flight-related software is not excluded from modernization.

Our process begins by upgrading the existing Fortran application to standard conforming Fortran 90/95. Next, interfaces to the original application routines are introduced to add safety features by detecting common programming errors. These interfaces ensure that the wrapper layer, added next, always correctly calls the legacy code. This wrapper layer allows problem based object abstractions to be introduced that interact cleanly with the legacy code, while supporting new enhancements. It also preserves the original, mostly unmodified, legacy software. The user can communicate with the modernized code across these layers and continuous development can occur simultaneously among these layers. The figure below shows the general approach



2.2 Fortran 90/95 Features that Modernize Programming

Many new features in the Fortran 90/95 standard provide benefits that are unfamiliar to experienced Fortran 77 software engineers. These features add safety, simplify complex operations, and allow software to be organized in a logically related way. Since backward compatibility is preserved one can incrementally make modifications while preserving existing work. Briefly, some of these new features are:

Modules Encapsulates (groups together) data, routines, and type declarations while providing accessibility across program units.

```
MODULE module_name
   implicit none
   save
   ! constant and variable declarations…
CONTAINS
   ! member subroutines and functions…
END MODULE
```

Use-Association Controls access to module content across program units.

```
PROGRAM program_name
   use module_name
   implicit none
   ! program content…
END program_name
```

Interfaces Verifies that the argument types in the procedure call match the types in the procedure declaration.

```
PROGRAM program_name
   implicit none
   interface
         subroutine foo(arg1, arg2)
         real, dimension(10,10) :: arg1
         logical :: arg2
         end subroutine
   end interface
END program_name
```

Derived Types User-defined types that support abstractions in programming. The creation of these types allows one to support problem domain based design.

```
TYPE new_type
   integer :: component_1
   real :: component_2
END TYPE new_type
```

Array Syntax This syntax simplifies whole array, and array subset, operations.

```
integer, dimension(100) :: x, y, z
   x = y * z                     ! Fortran 90/95 style…
   do i = 1, 100                 ! Fortran 77 style…
       x(i) = y(i) * z(i)
   end do
```

Dynamics Various kinds of dynamic structures are supported including allocatable arrays and pointers.

```
real, dimension(:), allocatable, save :: h
real, dimension(:), pointer :: p, q
  allocate(h(10), p(10))    ! Perform runtime allocation…
  q => p                    ! Refer to dynamic variable…
```

A very powerful realization is that combining these ideas allows support for object-oriented concepts. There are a number of textbooks on Fortran 90. One that we recommend is “Fortran 90 Programming”, by Ellis, Philips, and Lahey, Addison Wesley, 1994.

2.3 The Emergence of Software Tools

As the importance of re-engineering becomes more widespread companies have been more active in tool development. While our re-engineering process is not automatic it is largely mechanical so some aspects of our approach could benefit from software tools. Some companies claim to have tools that perform modernization from Fortran 77 to Fortran 90/95 automatically. An example is Simulog’s Foresys tool (http://www.simulog.fr/iforef.htm). Unfortunately, such tools can restructure in a difficult to comprehend manner (variable names are changed), do not add abstraction, and often leave software in a form nearly impossible to modify or extend.

Scitools’ Understand for FORTRAN (http://www.scitools.com/uf.html) “is an interactive development environment (IDE) tool providing reverse engineering, automatic documentation, metrics and cross referencing of FORTRAN source code…. [It helps you] reverse engineer and understand large amounts of legacy FORTRAN source code…. It also includes numerous graphical reverse engineering views designed to help you understand and assess changes you are considering in your code”.

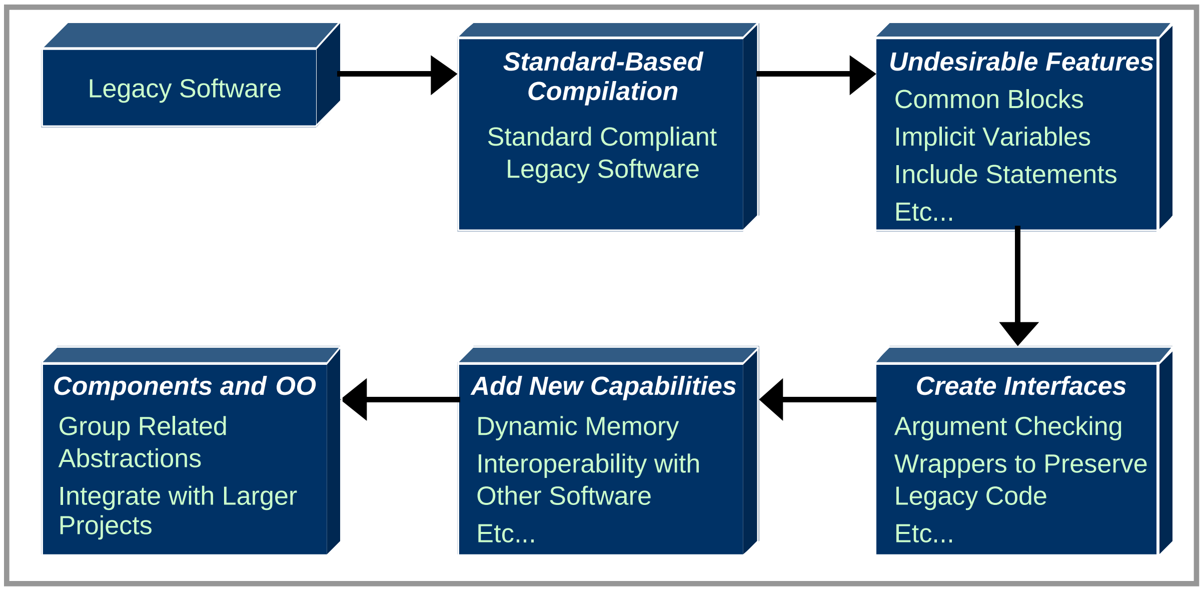
Pacific-Sierra Research (http://www.psrv.com/vast77to90.html) offers VAST/77to90 that allows you to “Automatically move your older Fortran programs to the new standard, with many features and options. COMMONs are replaced with MODULEs, loops with array syntax, fixed format with free format, GOTOs with new structured control statements, and much more.”

Honeywell has a tool called called DOME (http://www.htc.honeywell.com/dome/) that “is an extensible system for graphically developing, analyzing and transforming models of systems and software”. This tool is used as part of Boeing’s Incremental Upgrade of Legacy Systems project that will be described later.

Quibus Enterprises offers Forwarn (http://www.fortran.com/quibus_forwarn.html) that “is a static analyzer and documentation generator for Fortran programs. It speeds program development and debugging, improves reliability and portability, and helps reverse engineering. Forwarn does for Fortran what Lint does for C, and also documents routine and variable usage. Forwarn can check for common coding errors, print cross-reference listings, and print a calling tree diagram”.

3 Step-by-Step Process for Legacy Software Modernization

In this section we describe the process that one can follow to modernize legacy software. This process has been successfully applied and it defines a plan of action for such projects.



The diagram shows the fundamental stages involved. While we focus on Fortran legacy codes the same stages could be modified for software written in other languages. Many of the specific actions taken will also depend on the code structure and objectives.

3.1 Clearly Identify the Objectives

It is very important to have a conversation with the software owners to determine their objectives. The flowchart of the modernization process may help guide this discussion.

3.2 Understand the Legacy Software

Understanding, even at a basic level, how the legacy software is organized is valuable. While it is possible to perform the modernization without detailed knowledge of the application, knowing the design is very helpful. Here are some common questions that should be asked.

- Is this a stand-alone code or is additional software required?
- Is this a single language code or a multilanguage code?
- What platforms are required?
- Who is responsible for answering questions if legacy bugs are detected?
- What kind of obsolete features exist in the software?
- Are any third-party developers involved and is their software proprietary?

3.3 Standard-Based Compilation

This stage involves recompilation of the application with a modern compiler. This finds errors that were previously undetected, identifies extensions that are not part of the standard, and resolves other issues. These may include implicit and/or duplicated variables and use of non-standard routines and constructs. If the software relies on pre-compiled binaries where the original source code is not available then it is important to verify that these binaries will link with the recompiled legacy code. This can often be the case if the legacy code depends on libraries for graphical output, mathematics, or other features. In general, however, these libraries can also be recompiled as part of this process. During this stage all compiler reported errors should be corrected before moving forward.

Many applications on UNIX systems rely on "make", a utility that reduces the compilation steps for programs consisting of many files to a single command. Since Fortran 90/95 has more features than Fortran 77 additional information must be provided during the compilation process when these features are used. For example, Fortran 90/95 compilers generate .mod files that are similar in nature to .o object files, but actually are quite different. It may be necessary to tell the compiler where these files are generated so be certain to check the options provided by your specific compiler. In many cases this takes the form:

```
   f90 sources.f90 –M<.mod_directory>
```

where the –M option specifies the directory where these temporary .mod files exist.

At this stage the legacy code is being modified. For this reason, and for the following stages, some form of version control is recommended. One popular system is the Concurrent Versions System (CVS) (http://www.cvshome.org/).

3.4 Addressing Undesirable Features

One of the most undesirable features in legacy Fortran 77 codes are `COMMON` blocks since they often inhibit more advanced features, like dynamic memory. They also discourage code sharing since everything is exposed. For this reason, modifying large common blocks can also be intimidating since inadvertent errors are easy to introduce. Other undesirable features include implicitly declared variables, which are dangerous, and include statements that are platform dependent based on how directories are specified.

Common blocks can be handled by placing the specification in a Fortran 90/95 module. Furthermore, rather than using `include` to make a textual substitution, the module information can become accessible using the Fortran 90/95 `use` statement in the appropriate routines.

```
! Original COMMON Block in common.inc
real arg1(10,10), arg2(10,10)
logical arg3
integer arg4
COMMON /BLOCK1/ arg1,arg2,arg3,arg4
SAVE /BLOCK1/

subroutine foo()              ! Defined in some file…
include 'common.inc'
…
end

! Modernized Version in common.f
MODULE common_block1
   implicit none
   save
   real, dimension(10,10) :: arg1, arg2
   logical :: arg3
   integer :: arg4
END MODULE common_block1

subroutine foo()              ! Defined in some file…
use common_block1
…
end subroutine foo
```

The structure of the replacement is straightforward. One could have simply copied the original common block from common.inc into a module exactly, but using the Fortran 90/95 constructs gives additional advantages. These include the ability to make the block members dynamic and the ability to add more functionality to the module by making other modules visible within its scope, to name a few.

Other important issues to consider are free-format versus fixed-format program structure and naming conventions. Most legacy Fortran 77 codes use a fixed-format style. This is supported by Fortran 90/95 compilers but we recommend using free-format style wherever possible to increase readability. When creating new files as part of the modernization process (creating common block modules for example) free format should be used. When making modifications to existing legacy files (replacing `include` of common blocks with `use` of modules) fixed-format may be preserved. Compilers support options that allow fixed-format and free-format files to be used in the same executable.

Choosing a convention for general purpose naming and formatting is important as new modules and files are created in the modernizing software. The previous example shows how the legacy names are mapped into the modified names.

Variables that are created implicitly can introduce bugs that are very hard to detect and correct so we recommend that `implicit none` be specified so that all data is declared before it is used. This allows the compiler to report improperly declared variables.

3.5 Creating Interfaces

Interfaces are very important, as they add safety to the modernized software. They allow the compiler to verify consistent argument usage for procedures, which allows subtle errors to be detected and corrected in legacy codes.

Interfaces are created automatically for routines that are defined within modules, but we are currently interested in building interfaces for the legacy routines that will not be moved into modules at this time. Not every legacy routine requires an interface, but all of the routines accessible from the main program should have an interface. Furthermore, any routines in the scope of the main program that have arguments that will be dynamic will require an interface.

The `interface` statement is used to declare the procedure name and the types of its arguments. Since this is a Fortran 90/95 construct that will tie in the legacy code to the modernized code a new Fortran 90/95 interface.f file can be created to declare the Fortran 77 legacy interfaces. These interfaces can be placed in a module that in turn may use other modules, such as the common block modules recently created.

```
! Interface Module in interface.f

MODULE interface_module
  USE common_block1
  implicit none
  save
  interface
       subroutine foof77(arg1, arg2, dim1)
       real arg1(dim1,dim1)
       integer dim1
       logical arg2
       end subroutine
  end interface
END MODULE interface_module
```

Note that the interface has exactly the same declaration as the original Fortran 77 legacy procedure, in fact it is best to just copy it explicitly. This means that when the legacy routine is called additional checks will be performed to ensure that the number and types of the arguments match exactly.

It may look like very little has been gained, but the benefit of the interface becomes clear when it is combined with a wrapper that allows more powerful Fortran 90/95 features to be applied. For example, many Fortran 77 programs have very long argument lists because extra information must be included, such as the dimension of arrays. Since Fortran 90/95 arrays know their size these arguments do not need to be included in a wrapper function that calls the original legacy procedure.

```
! Interface Module in interface.f

MODULE interface_module
   USE common_block1
   implicit none
   save
   interface
         subroutine foof77(arg1, arg2, dim1)
         real arg1(dim1,dim1)
         integer dim1
         logical arg2
         end subroutine
   end interface
CONTAINS
   subroutine foof90(arg1, arg2)                    ! Wrapper
   real, dimension(:,:) :: arg1
   logical :: arg2
         call foof77(arg1, size(arg1,1), arg2)      ! Legacy
   end subroutine foof90
END MODULE interface_module
```

This is a simple example, but the effect can be significant for very complex procedures. In fact, more functionality (such as dynamic memory) can be applied at this level using the wrapper while preserving the original legacy software. Furthermore, this can be achieved without a serious performance penalty when the legacy routine is non-trivial.

The interfaces can also clarify how Fortran 77 style arguments are sometimes passed to procedures. For example, it is not uncommon to find Fortran 77 programs that pass a two-dimensional array to a procedure that expects a one-dimensional array. This can cause compile errors when interfaces are used because they require that the arguments must match exactly. In such instances it is possible to create multiple interfaces to recognize this difference using a generic procedure to allow a single name to select the correct module procedure based on the argument list.

```
! Interface Module in interface.f
MODULE interface_module
   USE common_block1
   implicit none
   save
   interface
         subroutine foof77(arg1, arg2)  ! Legacy Interface
         real, arg1(*), arg2(*)
         end subroutine
   end interface

   interface w_foof90                    ! Generic Wrapper
         module procedure foof90_1, foof90_2
   end interface

CONTAINS

! Two identical arguments…
   subroutine foof90_1(arg1, arg2)
   real, dimension(:) :: arg1, arg2
         call foof77(arg1, arg2)         ! Legacy
   end subroutine foof90_1

! Two different arguments…
   subroutine foof90_2(arg1, arg2)
   real, dimension(:) :: arg1
   real, dimension(:,:) :: arg2
         call foof77(arg1, arg2)         ! Legacy
   end subroutine foof90_2
END MODULE interface_module
```

In the example above, calling the generic procedure `w_foof90` will refer to `foof90_1` if both arguments are one-dimensional arrays and to `foof90_2` if arg1 is a one-dimensional array and arg2 is a two-dimensional array. Note that the original `foof77` legacy procedure has not been modified.

3.6 Adding New Capabilities

Now that the interfaces have been created and wrappers have been introduced to encapsulate the legacy software new capabilities can be added. For most legacy software the most desirable feature is dynamic memory. Fortran 90/95 supports many kinds of allocatable structures and they are straightforward to use. Dynamic memory increases the flexibility of the software since this frees the application user from fixed problem sizes. Interoperability with more modern software can also be achieved since the wrappers can be designed to utilize such applications. These new capabilities can be added without affecting the use of existing systems.

```
! Legacy Fortran 77 include file of static COMMON data

parameter (mdttl=128)
integer nElt, RayID(mdttl,mdttl), …
COMMON /EltInt/ nElt, RayID,…
SAVE /EltInt/

! New Module for COMMON data

MODULE elt_common
   implicit none
   save
   integer :: nElt, mdttl = 128
   integer, allocatable, dimension(:,:) :: RayID
CONTAINS
! Constructor
   subroutine new_elt_common()
         allocate( RayID(mdttl,mdttl) )
   end subroutine new_elt_common
END MODULE elt_common

! Dynamic allocation from main program

PROGRAM example
   use elt_common
   implicit none
   call new_elt_common()
   …
END PROGRAM example
```

The example above shows a legacy Fortran 77 common block with static data can be reorganized to support dynamic memory. This occurs by moving the common block into a module and specifying which structures will be dynamic. A constructor can be created to perform the allocation of the dynamic structure and this constructor can be called from the main program. A number of additional safety features such as checking if the structure was already allocated, handling of exceptional conditions like insufficient memory, and so forth can be added as well.

3.7 Moving toward Components and Object-Oriented Design

Fortran 90/95 contains derived types, like structures in C, which allow users to create their own types. This allows one to program using designs that better represent the problem domain. One of the major benefits of the methodology is that one can incrementally evolve the legacy code toward such a design while preserving the functionality of the legacy software. An object-oriented design allows the implementation details to change without affecting the user. In a sense, the interfaces and wrappers have hidden the details of the legacy software, but we can enhance the wrappers to support derived types evolving the code toward an object-oriented, component-based, design.

```
! Creating derived types for object-based design

type species
   real, dimension(:,:), pointer :: coords
   real :: charge_to_mass, kinetic_energy
end type species

! Using a legacy routine through an OO wrapper

subroutine w_push(particles, force, dt)
type (species) :: particle
real, dimension(:) :: force
real :: dt, qbm, wke
integer :: ndim, nparticle, nx
   ndim = size(particle%coordinates,2)
   nx = size(force)
   qbm = particle%charge_to_mass
   wke = particle%kinetic_energy
   call push(particle%coords, force, qbm, wke, ndim,
               nparticle, nx, dt)
end subroutine w_push
```

This example shows a legacy `push(…)` routine, wrapped by a Fortran 90/95 `w_push(…)`, routine that uses a derived type to group together related information. This was not possible in Fortran 77 so long complicated argument lists were required. The species type has a dynamic component, and other information, which simplifies the programmer's view of the data. Nevertheless, the original legacy software can still provide the functionality required.

```
! Creating classes for object-based design

MODULE plasma_class
! Create Derived Types…
CONTAINS
   subroutine new_species(…) ! Constructor…
   subroutine w_push(…)      ! Class Members…
END MODULE plasma_class
```

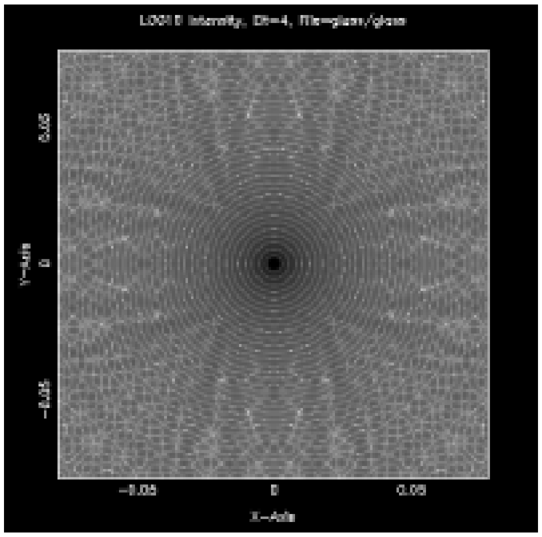
In fact, a class can be created which groups together operations common to the new species type where the class member routines utilize legacy software internally. New software can be added to the class as well. This is a very powerful concept, but careful planning is always required when building an object-oriented design.

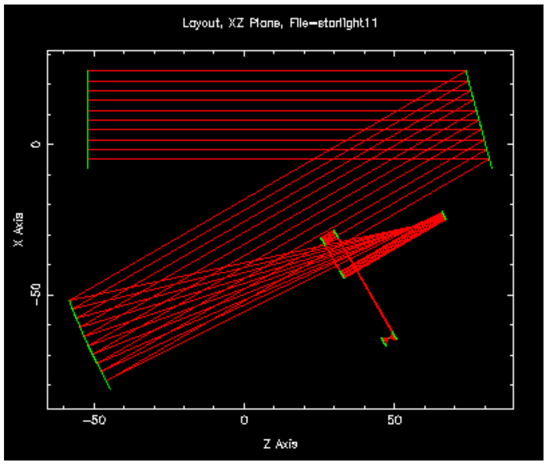
Once the classes have been designed, and tested, the modules can be incorporated into the main program and calls to the member routines can replace calls to the original legacy software. Since interfaces for the legacy software still exist this process can be incremental, the software still works at the end of the day, and development can continue during the modernization process allowing existing objectives to be satisfied.

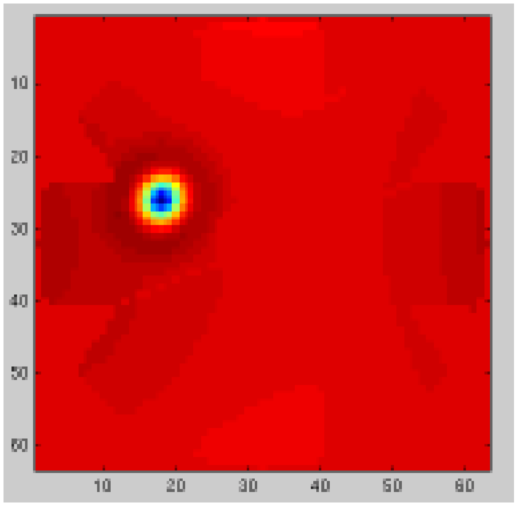
4 MACOS Case Study

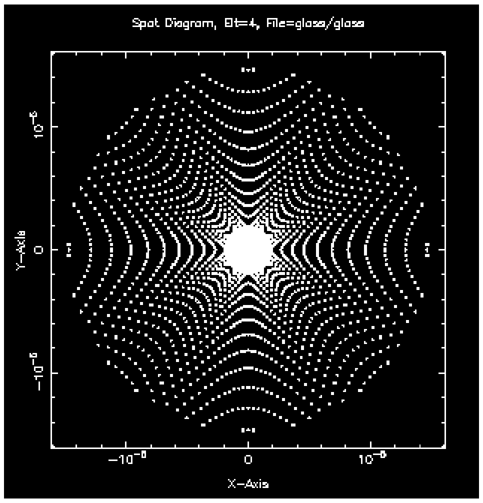
The Modeling and Analysis for Controlled Optical Systems (MACOS) software is an important NASA code that has been used for numerous projects. This software, developed by Dr. David Redding and others from the Optical Systems Modeling Group (385), provides powerful optical analysis tools and a unique capability for system-level design and analysis tasks. MACOS has many features, but a short list includes:

- Modeling optics on dynamic structures, deformable optics, and controlled optics
- Efficient general ray-trace capabilities
- Integrated support with other tools to create an end-to-end instrument system model









MACOS is written primarily in Fortran 77 and it interoperates with Matlab, PGPlot, and FFTw. There is also a subroutine library called SMACOS based on MACOS. Previous efforts to rewrite the software completely in C++ (to meet new objectives) were abandoned primarily because the new code did not perform as desired, and the designers are more fluent in Fortran.

The objectives of the designers were to achieve Fortran 90/95 standard compliance, dynamic memory support, and to hopefully have some subtle bugs corrected during the process. Reorganization for an object-oriented design was also of interest, but not a near-term requirement as more capabilities were being added to MACOS and SMACOS during the modernization process. It was important that the software remain in use during this effort.

4.1 Legacy Software Characteristics

MACOS and SMACOS version 2.8 is distributed across approximately 60 files consisting of nearly 67,200 lines of Fortran code (mainly Fortran 77). The software uses FFTw for Fourier Transforms and Matlab for visualization and as a calling program for SMACOS. There are approximately 765 procedures in the source code. It is well organized and runs under UNIX on Sun workstations as well as the Apple Macintosh. Most of the software was developed at JPL, but some features have been added by other organizations. It has been used to win numerous JPL flight projects. The primary developers were available and willing to respond to questions and issues during the modernization process, but they were not involved in the programming.

4.2 Modernized Software Characteristics

The modernized software added approximately 3,500 additional lines of Fortran 90/95 to the legacy code. Migrating to a Fortran 90/95 standard compliant version consisted of removing the machine specific `byte` data type that was no longer used and resolving issues detected by the compiler, such as improper type declarations, but these were minor. Most of the new code consisted of wrappers and interfaces created to preserve the legacy routines accessible from the main program. About 28 new wrappers/interfaces also allowed for dynamic memory to be added.

10 new modules were formed, mainly to replace existing `common` blocks. These modules were used about 540 times in subroutines, and other modules, to replace common block include statements. Additional changes were made to the Matlab configuration files so that the Fortran 90/95 version of SMACOS would function properly. The PGPLOT plotting package was also recompiled.

In addition to adding the new features requested there were also opportunities to replace obsolete legacy features. These included inconsistent types in argument passing to procedures, the use of `common` for temporary storage, and removal of non-portable code. There were very few legacy bugs, but they did cause system dependent problems at times.

The process found cases of inconsistent character lengths passed among procedures and instances where the names in declarations and the associated arguments to procedures did not match.

We spent about 1/2 a work year on the modernization, but we estimate that it could have been completed 2 to 3 times more quickly if the original authors were more directly involved. A significant amount of time was spent becoming familiar with the design and organization of the application. The process improved the software and it always produced the proper result at the end of the day. The incremental approach of the methodology was very beneficial since detailed understanding of the application itself was not required to meet the project objectives.

5 Analysis of Selected Flight Software

Part of our work involved investigating various mission flight, data, and analysis systems projects to determine the importance of legacy software issues in meeting project goals. We held meetings with representatives of the Mission Data System project, Boeing Phantom Works, the Navigation and Flight Mechanics Section (Peter Breckheimer), and the Mission Software Systems Section (Roger Lee).

5.1 Mission Data System

The Mission Data System (MDS) is an effort to introduce improved software that moves away from mission specific products to ones where software reuse and collaboration across the entire mission development cycle are emphasized. During our discussions with Bob Rasmussen and Dan Dvorak we learned that much of the MDS legacy software is written in C while new software development is primarily in C++ and JAVA. The project, at this time, consists of a few hundred thousand lines of code, but much more is expected. The main interest in preserving software would be for the ground systems software. We also learned that a software modernization effort was underway in the Navigation Group.

They wanted to know if our efforts were limited to scientific/numerical codes and/or necessarily Fortran legacy codes. Although our best experience is in that area the methodology is not language specific. They consider the ability to update a code incrementally important, including the ability to transfer this knowledge to developers.

We also met with Bob Toaz who was very much interested in the work, but it was not directly applicable to him since he is more involved in science data processing for MDS. He did mention that there are a number of legacy issues related to linking the newer flight software to the ground legacy software.

Rick Borgen also gave his views on modernization issues and legacy software related to MDS. He described how issues with MDS involve an interaction between new design, support of legacy systems, and development support for large systems. He is very experienced with issues related to the ground systems and cataloging. We had an informal discussion on the topic of software development for large systems and the issues one must consider when designing a new approach. For MDS this involves developing new software from scratch using C++/JAVA and also supporting systems developed in C and other languages. An important part of maintaining such systems is to have automated regression testing since, as code is modified, it can be difficult for developers to feel confident in the changes that occur. When considering the lifecycle of development from excitement with large funds at the start of a project, moving toward ultimate support by one person who is not completely familiar with the project, having a testing system that validates changes removes the fear of breaking the code. Ultimately, people are hesitant about modifying large projects because of the fear that the software may become unusable.

With MDS, some aspects of development are so new that preserving legacy systems is not useful. In other areas, such as the ground system, preservation is very important. For the ground system the idea of interfaces is useful since it allows the new communications systems on the spacecraft, and at the end-user, to work within the legacy ground system. For instance, in existing spacecraft the telemetry data is an almost instantaneous view but in future systems smart objects can be queried for more sophisticated information like “what occurred over the last day”. Sending this information over the ground system may be very different from the way it appears when it is created and received.

In Rick’s experience with MDS there are clearly areas where legacy issues have importance and where they do not. Also, there are times where one must just accept what exists and ensure that it can be used successfully without complaint. For example, if the processor on the spacecraft is not very sophisticated (perhaps it is radiation hardened) it is generally not possible to change it after launch. The expertise with this processor must be maintained even though one may wish something more advanced was available.

5.2 Boeing Phantom Works

Boeing Phantom Works in St. Louis has an Incremental Upgrade of Legacy Systems (IULS) Program, developed under the U.S. Air Force, that uses a wrapping technique to combine legacy and new avionics software for embedded platforms. As part of their “Bold Stroke” program they are trying to develop a tool set for automated wrapper development for legacy systems. The legacy codes are written primarily in Ada, Assembly language, CMS (used by Navy computing systems) and Jovial (a Fortran-like military language for embedded software). There is also development in C and C++. Their tool has been demonstrated both in the laboratory and in flight on the F15E fighter’s Overload Warning System. This system consists of about 250,000 lines of legacy Ada code while the upgraded flight software was generated in C++. Another laboratory demonstration rehosted the C-17 communications control unit software from a legacy military standard processor to a commercial processor where the legacy Jovial software was never modified. Dr. David Corman is head of the IULS project which began in 1996.

Boeing faces many of the same concerns that we have identified and they recognize the benefits of wrapping legacy software. The issues of importance to them are the following:

- New engineers are unfamiliar with the old software
- Moving toward reusable components is desirable
- The need to reduce risk from re-engineering is critical
- The legacy software is tested and certified
- Upgrading software, incrementally, without a complete re-engineering effort

Additional goals are migration of their codes toward open system architectures and moving toward a common framework for plug-and-play design in new architectures. The underlying legacy software continues to evolve and their primary wrapper toolset is based on producing C++ code.

Boeing collaborated with Honeywell in the development of a domain engineering tool called WrapidH. This is a graphical tool supporting automated wrapper development for legacy software. The user graphically inputs the legacy software and develops a new architecture specifying data relationships among the legacy and modern code. The key feature of the tool, in our opinion, is the ability to construct interfaces within the software for data transfer. When the interfaces are specified accurately this allows a library of reusable parts to be defined. One goal of the tool is to extend it toward the development of reusable components. WrapidH runs on PC/Windows and NT environments. Honeywell was the lead developer and General Dynamics played a large role in the documentation effort. This process is largely, but not completely, automated.

There are three general approaches used in the IULS wrapper strategy:

Rehost	Recompilation and driver fixes are applied when a processor is upgraded.
Hybrid	Some legacy functionality is retained and new software is used as needed. The wrapper acts as a bridge between the old and new systems.
Emulate	The old instruction set is emulated on a new processor since the original software function is stable and may not change.

The wrapper can be multilingual and the may add new functionality. They have even used wrappers to set up run-time environments for legacy systems. The IULS team consists of 2-3 people and a number of subcontractors.

In the future, Boeing wants to transition their toolset to different Air Force programs. They expect that software rewrites could be reduced from years to months. Also, as they transition to open architectures they expect that a common set of software could be moved easily among many kinds of aircraft. Boeing recently agreed to release the user manual for WrapidH to JPL allowing us to gain a better understanding of how their technology works. They are very interested in endorsement of their software and technology transfer.

5.3 Navigation and Flight Mechanics (Section 312)

The Navigation Software Development Group has decided to transition their software from Sftran/Fortran 77 to Fortran 90/95. Although their original motivation was Hewlett-Packard’s phasing out of support for Fortran 77 they now see an opportunity to re-engineering aspects of their software. At Peter Breckheimer’s (312) request, Viktor Decyk (385), Jack Hatfield (368), Charles Norton (385), and Van Snyder (327) gave a two-day seminar on the new features of Fortran 90/95 that support better organization and safety for large scale programming.

This group supports a number of projects and has concerns about meeting deadlines and ensuring that the accuracy/functionality of their application is preserved, especially if the software is completely rewritten. Much of our conversation and seminar focused on how legacy software can be preserved, while new functionality can be added, based on our modernization methodology. The seminar was also educational since we focused on the most appropriate new features in Fortran 90/95 that should help the software engineers get through personal study using textbooks more quickly.

The software under consideration consists of over 6 million lines of Fortran, where a large portion is generated by the Sftran/Fortran 77 tool. There is an interest in moving away from this tool since another tool supports conversion from Sftran/Fortran 77 to Fortran 90, but it could use further evaluation. The key concerns of this group are the ability to have tools available to help with the re-engineering, to ensure that they can always meet mission deadlines, and to preserve the detailed knowledge specific to their application. For these reasons, they find our methodology as a potential benefit so we expect to pursue a relationship with this group.

5.4 Mission Software Systems (Section 369)

Roger Lee (369) gave an overview of a project involving modernization of DSN ground software led by Laverne Hall (369), who is in his section. The technology regarding the sensitivity limits of the antennas is near optimal, but the software developed for this was written many years ago. New programming methods, and even languages, exist and this opens the opportunity for new functionality to be added to this system.

Speaking in general about the opportunities for software modernization around JPL we discussed how advances in spacecraft processors are driving new opportunities to perform more onboard processing. For some projects, including parts of MDS, this implies that rewriting may be more beneficial than preserving some legacy software since it is designed for processors of lesser functionality. He also mentioned that Elaine Shell of GSFC is involved in software modernization efforts and that she was another potential source of information.

6 Experiences and Comments

Our interaction with CSMISS allowed us to validate our methodology, and it has even promoted new work. The new MACOS software has been successfully delivered and the project is now interested in pursuing new directions. This includes adding object-oriented concepts and added enhancements. We expect to apply this methodology to the navigation software as they transition from the legacy software to their next generation software.

We did not use any software tools in applying our methodology; everything was done by hand. At this point, we will examine various tools that could be useful for partial automation of this methodology.

Legacy software still has great value, but extending that functionality has become more important. Modern applications require greater complexity and support for multiple authors. More flexible design and dynamic features are also beneficial. Our methodology allows a modern superstructure to be erected around a legacy code where data abstraction and information hiding help to limit exposure of unnecessary details. Furthermore, modern software features, combined with improvements in compilers, help to reduce inadvertent errors. Applying wrappers to protect legacy software allows one to extend the functionality of that software. The re-engineering methodology’s cost effectiveness, speed, and ability to protect one’s investment in the experience and knowledge embedded within legacy software should be beneficial to a variety of mission software projects.